Lab 3 More

Memory Management

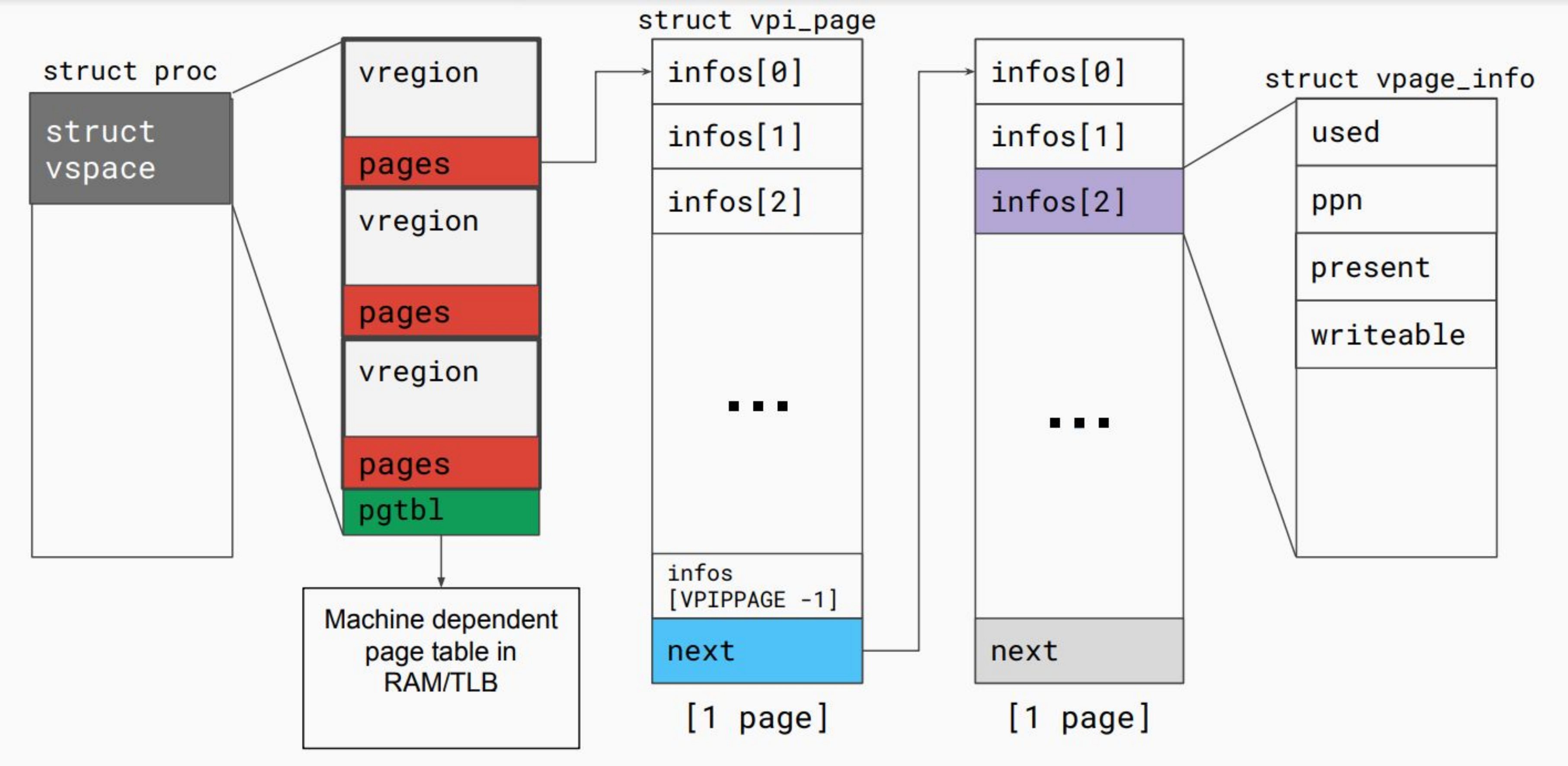
Reminder

- Lab 3 design doc is due tonight! 2/15/24
- Lab 3 Code due 2/23/24
- Pset 3 Out Tomorrow! 2/16/24
  - Due 2/23/24

Today’s Agenda

- More detail on vspace and vspace functions
- Some discussion questions on lab 3
- Q&A time/ Open OH

vspace Visual Diagram



Vregions vs Page Tables

- What's the difference between vregions/vpage_infos and the page table?
- Can you make modifications to struct vpage_info?
- What happens if you make changes to vregions/vpage_info? Is it automatically reflected on the page table?

```
struct vregion {
  enum vr_direction dir;    // direction of growth
  uint64_t va_base;         // base of the region
  uint64_t size;            // size of region in bytes
  struct vpi_page *pages;   // pointer to array of page_infos
};
```

region metadata

```
struct vpage_info {
  short used;      // whether the page is in use
  uint64_t ppn;    // physical page number
  short present;   // whether the page is in physical memory
  short writable;  // does the page have write permissions
  // user defined fields

};
```

page metadata

Vspace Functions

- Given a virtual address, how do you find which vregion it belongs to?
- Given a virtual address, how do you find its metadata (vpage_info)?
- How do you add new page to frame mapping?
- How do you update the page table to reflect changes in vregion/vpage_info?
- How do you flush the TLB?
- When would you want to flush the TLB?
- Do you need to flush the TLB after a new mapping is added?

Physical Memory Management

- our QEMU instance emulates 16MB of physical memory
- it is entirely mapped into the kernel virtual address range starting at KERNBASE
  - can easily find the physical address backing a kernel virtual address: subtract KERNBASE from va
  - can the same thing be done on user virtual address?

```
#define V2P(a) (((uint64_t)(a)) - KERNBASE)
#define P2V(a) (((void *)(a)) + KERNBASE)
```

Physical Memory Management

```
struct kmap {
  void *virt;
  uint64_t phys_start;
  uint64_t phys_end;
  int perm;
} kmap[] = {
  { (void*)KERNBASE, 0,              EXTMEM,    PTE_W}, // I/O space
  { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},     // kern text+rodata
  { (void*)data,     V2P(data),     (uint64_t) npages * PGSIZE,   PTE_W}, // kern data+memory
  { (void*)DEVSPACE, 0xFE000000,    0x100000000,           PTE_W}, // more devices
};
```

KERNBASE = start of kernel address range
KERNLINK = start of kernel code
data = start of kernel data and heap
npage = total # of physical pages (frames)

^ what's being mapped in the kernel page table, also mapped into every page table

physical memory layout

device space

mapped to [data, data + rest of DRAM)

mapped to [KERNLINK, data)
data = start of kernel heap

mapped to
[KERNBASE, KERNLINK)

KERNLINK = KERNBASE + EXTMEM

```
+------------------+  <- 0xFFFFFFFFFFFFFFFF (18 exabytes)
|                  |
|      Unused      |
|                  |
+------------------+  <- 0x0000000100000000 (4GB)
|      32-bit      |
|  memory mapped   |
|     devices      |
+------------------+  <- 0x00000000FE000000 (4GB - 32MB)
|                  |
|      Unused      |
|                  |
+------------------+  <- depends on amount of RAM
|                  |
|                  |
| Extended Memory  |
|                  |
|                  |
+------------------+  <- 0x0000000000100000 (1MB)
|     BIOS ROM     |
+------------------+  <- 0x00000000000F0000 (960KB)
|  16-bit devices, |
|  expansion ROMs  |
+------------------+  <- 0x00000000000C0000 (768KB)
|   VGA Display    |
+------------------+  <- 0x00000000000A0000 (640KB)
|                  |
|    Low Memory    |
|                  |
+------------------+  <- 0x0000000000000000
```

Physical Memory Allocation

- `kalloc` allocates a physical frame, it returns the kernel page mapped to the physical frame for ease of access `return P2V(page2pa(&core_map[i]));`
- multiple system calls/kernel functions may call `kalloc` concurrently, what does `kalloc` do to keep these accesses safe?
- how does `kalloc` find a free frame?
  - by looking through metadata for frames (core_map)

```
struct core_map_entry {
  int available;
  short user;    // 0 if kernel allocated memory, otherwise is user
  uint64_t va;   // if it is used by kernel only, this field is 0
};
```

frame metadata

Core_map_entry

- Access should be protected by the kmem.lock
- Can add to the struct to track additional information (refcounts)
  - Why do we care about refcount?
  - When will the refcount be greater than 1?

```
struct core_map_entry {
  int available;
  short user;   // 0 if kernel allocated memory, otherwise is user
  uint64_t va;  // if it is used by kernel only, this field is 0
};
```

frame metadata

Page Faults Error Code

- Last 3 bits of tf->err
  - B2 is set if fault occurred in user mode
  - B1 is set if fault occurred on a write
  - B0 is set if the faulting page has a valid mapping to a physical frame

The Page Fault sets an error code:

```
 31              15                              4               0
+---+--  --+---+-----+---+--  --+---+----+----+---+---+---+---+---+
|   Reserved   | SGX |   Reserved    | SS | PK | I | R | U | W | P |
+---+--  --+---+-----+---+--  --+---+----+----+---+---+---+---+---+
```

	Length	Name	Description
P	1 bit	Present	When set, the page fault was caused by a page-protection violation. When not set, it was caused by a non-present page.
W	1 bit	Write	When set, the page fault was caused by a write access. When not set, it was caused by a read access.
U	1 bit	User	When set, the page fault was caused while CPL = 3. This does not necessarily mean that the page fault was a privilege violation.

Meaning of the bits

- When B0 (present bit) is set, what does this imply?
  - page fault not caused by lack of page to frame mapping!
  - must be a permission (page protection) error
  - when a stack growth (access to stack for the first time) occurs, will this bit be set?
  - when a write is done on a cow page, will this bit be set?
  - when a write is done on a mapped read only page, will this bit be set?

Meaning of the bits

- When B1 (read/write bit) is set, what does this imply?
  - access is a write
  - if we read on an unallocated stack page, will this bit be set?
  - if we write on an unallocated stack page, will this bit be set?
  - upon a cow read access, will this bit be set?
  - upon a cow write access, will this bit be set?

Meaning of the bits

- When B2 (user/supervisor bit) is set, what does this imply?
  - access is done from user mode
  - when a stack growth occurs, is this bit set? (can stack growth happen in kernel mode?)
  - when a cow fork occurs, is this bit set? (can cow happen in kernel mode?)

Copy-on-write Fork FAQ

- How do we keep track of physical pages and refcounts?
    - Everyone take a look at kalloc.c!
- What vspace function to write to support COW fork?
    - Which function do we currently use to copy? What should we replace it with? (Not a trick question, look in the spec.)
- What do the fields of a page (struct `vpage_info`) need to be after a copy-on-write fork?
        - How do you know if a given page is in use? How do you know it can be written to? How can you uniquely identify a page? How do you know which physical page the vpace maps to?
- What happens to a page that is already read-only before COW fork?

More COW

- Synchronization in modifying the **vspace** in page fault in COW fork?
  - Not needed -- current process has exclusive access to its own vspace (no multithreading)
  - **However, the ref count on the physical page could be concurrently modified**
- What can happen if a copy-on-write fork is not synchronized?

Helper Macros and Functions

P2V: physical addr to virtual addr

V2P: virtual addr to physical addr

PGNUM: physical addr to page number

va2vpage_info: virtual addr to vpi_info

Any questions?